Capitulo 1
Tipos de dados e Interfaces

Introdução

O .NET Framework 2.0 fornece uma porção de tipos predefinidos que são necessários para a criação dos mais diversos tipos de aplicações que o mesmo fornece para nós desenvolvedores. Esses tipos são utilizados constantemente nas aplicações, para armazenar valores, bem como uma estrutura extensível para que você crie seus próprios tipos.

Neste capítulo você entenderá como funciona a arquitetura de tipos do .NET Framework 2.0, verá também sobre os tipos intrínsicos disponíveis. Além disso, vamos abordar um assunto fortemente ligado aos tipos: tipos-valor e tipos-referência e, para complementar, analisaremos o boxing e unboxing, que é um grande vilão em termos de performance das aplicações .NET-based. Para finalizar a parte sobre tipos, temos ainda alguns tipos especiais que precisam ser examinados: *Generics*, *Nullable Types*, *Exceptions* e *Attributes*.

Na segunda parte do capítulo, vamos analisar as *Interfaces* que são disponíveis dentro do .NET Framework 2.0 e também veremos como criar suas próprias Interfaces para utilizar no seu código ou, se desejar, expor esta para que consumidores de seu componente consigam implementá-la, criando uma espécie de "contrato" que teus componentes exigem para poderem trabalhar.

Examinando a arquitetura e os tipos

CTS – Common Type System

Como todos sabem, os tipos são peça fundamental em qualquer tipo de linguagem de programação, seja para a criação de componentes ou para tipos de aplicações que interajam com o usuário (Windows ou Web). Como a idéia da Microsoft é tornar tudo o mais integrado possível, ela criou uma especificação de tipos chamada Common Type System (CTS), que descreve como os tipos são definidos e como se comportam.

Esses tipos são compartilhados entre todas as linguagens .NET. Sendo assim, você pode criar um componente em Visual C# e consumir este mesmo componente em uma aplicação cliente escrita em Visual Basic .NET sem nenhum problema, já que o tipo esperado ou o tipo a ser devolvido são de conhecimento de ambas. Você também ouvirá o termo *cross-language integration* para essa mesma explicação.

Além disso, um outro ponto importante do CTS é que também específica as regras de visibilidade de tipos para acesso aos membros do mesmo. Atualmente temos os seguintes modificadores de acesso disponíveis:

Modificador VB.NET	Modificador C#	Descrição
Private	private	Pode ser acessado por outros métodos somente dentro do mesmo tipo (classe).
Protected	protected	Pode ser acessado por outros métodos dentro do mesmo tipo e também por tipos derivados.
Friend	internal	Pode ser chamado de qualquer local desde que seja dentro do mesmo Assembly.
Friend Protected	internal protected	Pode ser acessado por outros métodos dentro do mesmo tipo e também por tipos derivados dentro do mesmo Assembly.
Public	public	Pode ser chamado de qualquer local.

Para finalizar essa introdução sobre o CTS, há ainda uma das principais e mais importantes regras definidas por ele: todos os tipos devem herdar direta ou indiretamente de um tipo predefinido: **System.Object**. Este é a raiz de todos os tipos dentro do .NET Framework e, conseqüentemente, será também a raiz para os tipos customizados por você dentro da sua aplicação e, sendo assim, ele fornece um conjunto mínimo de comportamentos:

- Representação em uma string do estado do objeto
- Consultar o tipo verdadeiro da instância
- Extrair o código *hash* para a instância
- Comparar duas instâncias quanto à igualdade
- Realizar uma cópia da instância

CLS – Common Language Specification

Graças à um ambiente de execução comum e informações em metadados, a CLR (Common Language Runtime) integra as linguagem e permitem que estas compartilhem os tipos criados em uma linguagem sejam tratado de forma igual na outra.

Essa integração é fantástica, mas muitas linguagens são bem diferentes entre si, ou seja, algumas fazem distinção entre maiúsculas e minúsculas, outras não oferecem sobrecargas de operadores, etc.. Se alguém desejar criar um determinado tipo que seja compatível com qualquer outra linguagem .NET, você terá que utilizar somente os recursos que obrigatoriamente estejam também nas outras linguagens. Para isso, a Microsoft criou o Common Language Specification (CLS), que especifica o conjunto mínimo de recursos que devem ser suportados se desejam gerar código para o CLR. Geralmente empresas que estão criando seus compiladores para .NET, devem seguir rigorosamente estas especificações.

Para assegurar que o componente que está desenvolvendo seja compatível com qualquer linguagem .NET, você pode incluir um atributo chmado *CLSCompliant* no arquivo *AssemblyInfo* que instrui o compilador a se certificar que um tipo público que está sendo

disponibilizado não contenha nenhuma construção que evite de ser acessado a partir de outra linguagem. O código abaixo exemplifica o uso deste atributo:

```
VB.NET
<Assembly:CLSCompliant(True)>

C#
[Assembly:CLSCompliant(true)]
```

Abaixo, a imagem ilustra o conjunto de tudo que vimos até o momento:

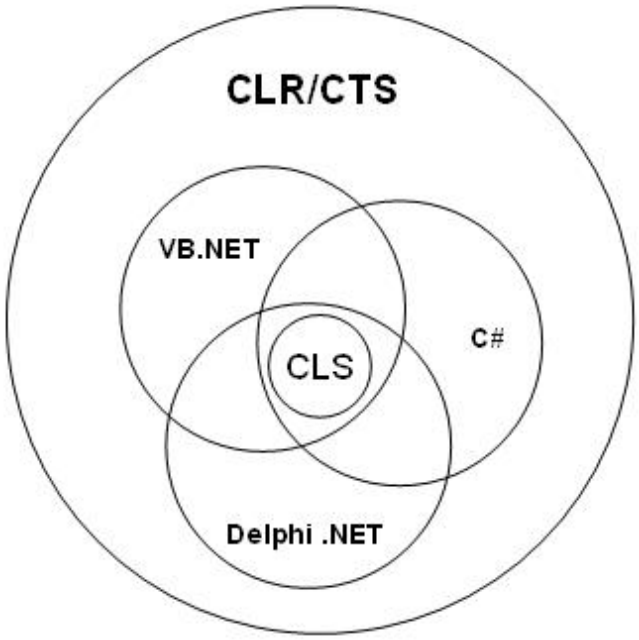

Imagem 1.1 – Uma exibição em forma de conjunto para entermos a relação entre elas.

Tipos fornecidos pelo .NET Framework

Dentro do .NET Framework 2.0, várias funcionalidades são encapsuladas dentro de objetos e estes, por sua vez, são instâncias de tipos fornecidos pelo sistema. O .NET Framework já traz internamente vários tipos predefinidos, quais são conhecidos como tipos básicos, ou *base system types*. Um exemplo típico disso é um controle *Button*. Internamente ele possui uma porção de membros que expõem ou recebem tipos de dados como strings, inteiros, decimais ou até mesmo outros tipos/objetos.

Há alguns tipos de dados que são muito comuns em várias linguagens de programação. Podemos citar vários deles: inteiros, strings, doubles, etc.. São tão comuns que grande parte dos compiladores permitem utilizarmos um código que manipule esses tipos de

forma simples, um "atalho". Se esse atalho não existisse, teríamos que fazer algo mais ou menos como:

```
VB.NET
Dim codigo As New System.Int32()

C#
System.Int32 codigo = new System.Int32();
```

Apesar de funcionar sem nenhum problema, isso não é nada cômodo, justamente pela freqüencia com qual utilizamos esses tipos. Felizmente tanto o Visual Basic .NET quanto o Visual C# fornecem várias keywords que o compilador faz o trabalho de mapear diretamente para o tipo correspondente dentro do .NET Framework Class Library (FCL). Esses tipos de dados que o compilador suporta "diretamente" são chamados de tipos primitivos. Com isso, o código acima ficaria muito mais produtivo se utilizássemos uma sintaxe semelhante a mostrada logo abaixo:

```
VB.NET
Dim codigo As Integer

C#
int codigo = 0;
```

Tipos valor e tipos referência

O Common Language Runtime (CLR) suporta duas espécies de tipos: tipos-valor (value types) e tipos-referência (reference types).

Os **tipos-valor** são variáveis que diretamente contém seu próprio valor, sendo assim, cada variável mantém a cópia de seus dados e, conseqüentemente, operações que utilizam essa variável não afetará o seu valor.

Tipos-valor são divididos em dois tipos: *built-in* e *user-defined types*. O primeiro trata-se de valores que são fornecidos com o próprio .NET Framework, como é o caso de inteiros, floats, etc.. Já o segundo, são tipos que definimos dentro de nossos componentes ou aplicações. Geralmente esses tipos são estruturas de dados ou enumeradores. As estruturas de dados são bem semelhantes a uma classe e podem conter membros para armazenar os dados e também funções para desempenhar algum trabalho. Já os enumeradores são constantes que fixam a escolha de algums dos valores fornecidos por ele, o que impossibilita o consumidor de passar algo diferente do que espera, ou melhor, de passar algum valor que seu método/classe não saiba trabalhar.

Já os **tipos-referência** contém ao invés do valor, uma referência para os dados e, sendo assim, é possível termos duas variáveis apontando para o mesmo lugar na memória, ou melhor, para o mesmo objeto. Como os tipos-referência apontam para o mesmo local, uma operação poderá alterar o conteúdo desta variável, já que o mesmo está "compartilhado". Geralmente todos os objetos dentro do .NET, sejam eles intrínsicos ao Framework ou os objetos customizados, são tipos-referência.

Os tipos-referência também são divididos em *built-in* e *user-defined types* assim como os tipos-valor. Para exemplificar, *built-in type* podemos a classe *Form* (Windows Forms) e a classe *Page* (Web Forms), entre muitos outros. Para *built-in types* temos os tipos/objetos que criamos para a aplicação/componente que estamos desenvolvendo. Apesar da imaginação ser o limite, temos alguns exemplos: *Cliente*, *LeitorCNAB*, etc.. A hierarquia dos tipos é ilustrada através da imagem 1.2:

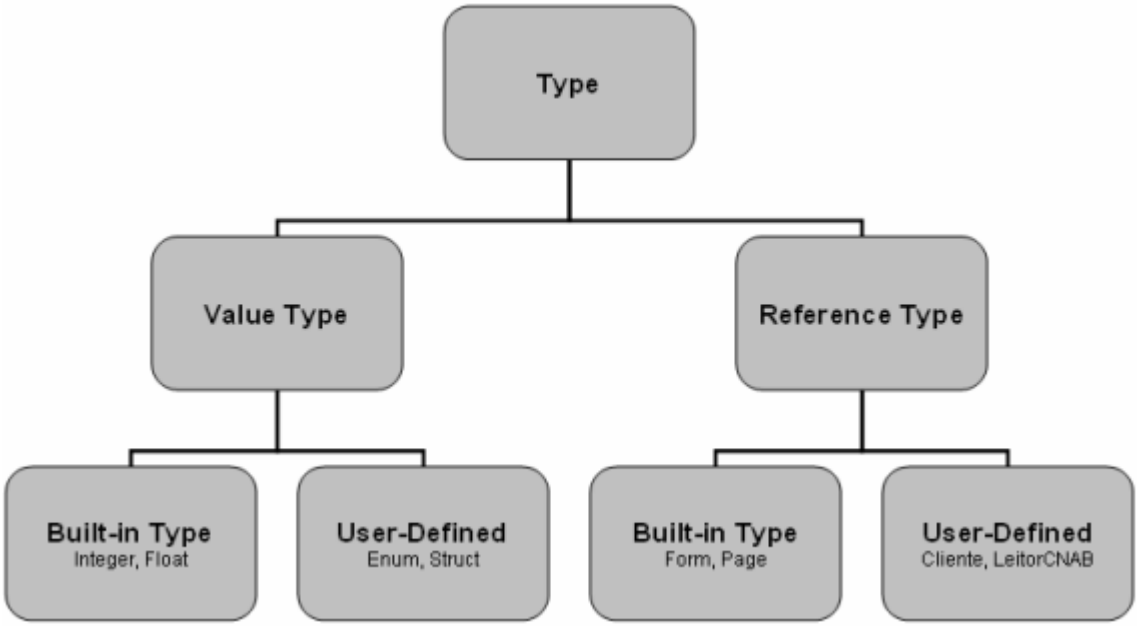

Imagem 1.2 – Hierarquia dos tipos do .NET Framework.

Além desta grande diferença entre os dois tipos, ainda há uma separação quando falamos em nível de memória. Cada um dos tipos são armazenados em regiões completamente diferentes. Os tipo-valor são armazenados na *Stack* e os tipo-referência na *Heap*.

A memória *Stack* é um bloco de memória alocado para cada programa em runtime. Durante a vida de execução, quando uma função é invocada, todas as variáveis que ela utiliza são colocadas na *Stack*. Assim que a função é retornada ou o escopo de um bloco é finalizado, o mais breve possível os valores ali colocados serão descartados, liberando assim, a memória que estava sendo ocupada. Como falamos anteriormente, quando uma variável do tipo-valor é passada de uma função, uma cópia do valor é passado e, se esta função alterar este valor, não refletirá no valor original. Como os tipos de dados dentro do .NET Framework são uma estrutura, eles são também armazenados na *Stack*.

No caso de tipos-referência, os dados são armazenados na memória *Heap*, enquanto a referência do mesmo é colocado na mémoria *Stack*. Isso acontece quando utilizando o operador *new* (*New* em VB.NET) no código, que retornará o endereço de memório do objeto. Isso acontece quando precisamos criar efetivamente o objeto para utilizá-lo. Para entender melhor esse processo, vamos analisar o código abaixo:

```
VB.NET
Dim cliente1 As Cliente()
Dim cliente2 As Cliente = cliente1

C#
Cliente cliente1 = new Cliente();
Cliente cliente2 = cliente1;
```

Quando atribuímos o *cliente1* ao *cliente2* recém criado, você está copiando a referência ao objeto que, por sua vez, aponta para uma determinada seção na memória *Heap*. No código acima, quando você efetuar uma mudança, seja ela no objeto *cliente1* ou *cliente2*, refletirá no mesmo objeto da memória *Heap*. Através da imagem 1.2, podemos visualizar as memórias (*Stack* e *Heap*) em conjunto quando armazenam tipos-referência:

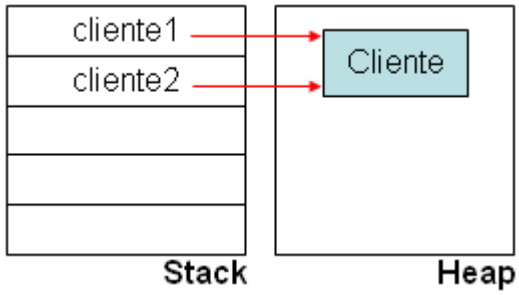

Imagem 1.3 – Memória *Stack* e *Heap* armazenando tipos-referência.

Como já falamos anteriormente, todas os objetos são tipos-referência. Além deles, ainda temos as *Interfaces* que também operam em modelo tipo-referência. As *Interfaces* contém métodos e propriedades em seu interior mas não possui nenhuma implementação concreta. Elas devem ser implementadas em classes para implementar a sua funcionalidade. As *Interfaces* serão discutidas mais detalhadamente ainda neste capítulo.

Nota Importante: Todos os tipos em .NET são, em última instância, derivados de **System.Object**. Como dito anteriormente, todos os tipos do .NET sejam eles intrínsicos ao Framework ou sejam objetos customizados, são tipos-referência. Mas e quanto aos tipos-valor? Não vimos que inteiros, floats são tipo-valor? Sim, mas a questão é que a Microsoft se preocupou com isso e criou uma possibilidade de diferenciar um de outro. Se analisar atentamente a documentação do .NET Framework, verá que a maioria dos tipos-valor, por exemplo *integer*, *decimal*, *datetime* são representados através de

estruturas de dados: *System.Int32*, *System.Decimal* e *System.DateTime* respectivamente. Toda e qualquer estrutura é derivada de **System.ValueType** e esta, por sua vez, herda de **System.Object**. Além da classe **System.ValueType** fornecer toda a estrutura básica para todos os tipos-valor do .NET ela também é tratada de forma diferente pelo runtime, já que seus tipos derivados devem ser colocados na memória *Stack*. Estrutura de dados e enumeradores herdam de **System.ValueType**.

Essa distinção entre tipos-valor e tipos-referência deve existir, pois o desempenho de uma aplicação poderia seriamente ser comprometida, pois imagine se a utilização de um simples inteiro, tivéssemos que alocar memória para isso. Os tipos valores são considerados "peso-leve" e, como vimos acima, são alocados na *Stack* da *Thread*. As melhorias em termos de performance não param por aí. Esses valores mais "voláteis" não ficam sob inspeção do *Garbage Collector* e fora do *Heap*, o que reduz o número de passagens do coletor.

Boxing e Unboxing

Como vimos na seção anterior, os tipos-valor são muito mais leves que tipos-referência porque não são alocados no *Heap* e, conseqüentemente, não sofrem coleta de lixo através do *Garbage Collector* e não são referenciados por ponteiros. Mas imagine uma situação onde você precisa obter uma referência a uma instância do tipo valor. Um caso típico para exemplificar são o uso de coleções. Há dentro do .NET Framework um objeto chamado **ArrayList** (*System.Collections*) que, permite-nos criar uma coleção de qualquer tipo. Como ele pode armazenar qualquer tipo, ele aceita nos parâmetros de seus métodos um **System.Object** (tipos-referência). Como tudo em .NET é, direta ou indiretamente, um tipo de **System.Object**, posso adicionar nesta coleção qualquer tipo, mesmo sendo um tipo-valor.

Mas e se quisermos adicionar um tipo-valor nesta coleção, como por exemplo, uma coleção de inteiros? Para isso, basta simplesmente fazermos:

```
VB.NET
Dim colecao As New ArrayList();
colecao.Add(1)
colecao.Add(2)

C#
ArrayList colecao = new ArrayList();
colecao.Add(1);
colecao.Add(2);
```


O método **Add** recebe um tipo-referência (**System.Object**). Isso quer dizer que o método Add exige uma refência (ponteiro) para um objeto no *Heap*. Mas no código acima, é

adicionanado um inteiro e tipo-valor não possui ponteiros para o *Heap*, já que são armazenados na *Stack*.

Isso só é possível graças ao *boxing*. *Boxing* na maioria dos casos ocorre implicitamente. Abaixo é o que acontece nos bastidores quando *boxing* ocorre:

1. A quantidade de memória é alocada no *Heap* de acordo com o tamanho do tipo-valor e qualquer *overhead* a mais, se for o caso.
2. Os campos do tipo-valor são copiados para a *Heap* que foi previamente alocada.
3. O endereço com a referência é retornado.

Felizmente compiladores das linguagens como Visual Basic .NET e Visual C# produzem automaticamente o código necessário para realizar o *boxing*. Isso é perfeitamente notável no código acima, pois não precisamos escrever nada mais do que precisaríamos no caso de adicionarmos um tipo-referência.

O *unboxing* é o contrário do *boxing*, mas considere sendo uma operação mais fácil em relação ao *boxing*. *Unboxing* é apenas obter o ponteiro para tipo-valor bruto contido dentro de um objeto e, neste caso, não é necessário copiar nenhum tipo de dado na memória. Obviamente que boxing e unboxing comprometem a performance da aplicação em termos de velocidade como de memória. Felizmente no .NET Framework 2.0, foi introduzido o conceito de *Generics Collections* que elimina completamente estes problemas. *Veremos sobre Generics Collections no Capítulo 2*.

Examinando tipos especiais

Generics

Generics é um novo conceito introduzido na versão 2.0 do .NET Framework. *Generics* permitem termos classes, métodos, propriedades, *delegates* e *Interfaces* que trabalhem com um tipo não especificado. Esse tipo "não especificado" quer dizer que estes membros trabalharão com os tipos que você especificar em sua construção. O melhor entendimento deste conceito provavelmente será quando estiver aprendendo sobre *Generics Collections*, que será abordado no Capítulo 2.

Como vimos anteriormente, o **ArrayList** permite adicionarmos qualquer tipo dentro dele. Mas e se quiséssemos apenas adicionar valores inteiros, ou somente strings? Isso não seria possível pois o método **Add** aceito um **System.Object**. Com *Generics*, é possível criar coleções de um determinado tipo, o que permitirá que o usuário (outro desenvolvedor) somente adicione variáveis do mesmo tipo e, se por acaso ele quiser adicionar algo incompatível, o erro já é detectado em *design-time*. O .NET Framework 2.0 fornece uma porção de coleções utilizando *Generics* que veremos mais detalhadamente no Capítulo 2 deste curso.

Classes genéricas oferecem várias vantagens, entre elas:

1. **Reusabilidade:** Um simples tipo genérico pode ser utilizado em diversos cenários. Um exemplo é uma classe que fornece um método para somar dois números. Só que estes números podem ser do tipo *Integer*, *Double* ou *Decimal*. Utilizando o conceito de *Generics*, não precisaríamos de *overloads* do método *Somar(...)*. Bastaria criar um único método com parâmetros genéricos e a especificação do tipo a ser somado fica a cargo do consumidor.
2. **Type-safety:** *Generics* fornecem uma melhor segurança, mais especificamente em coleções. Quando criamos uma coleção genérica e especificamos em seu tipo uma string, somente podemos adicionar strings, ao contrário do *ArrayList*.
3. **Performance:** Fornecem uma melhor performance, já que não há mais o *boxing* e *unboxing*. Além disso, o número de conversões cai drasticamente, já que tudo passa a trabalhar com um tipo especificado, o que evita transformarmos em outro tipo para termos acesso a suas propriedades, métodos e eventos.

Classes genéricas

Uma classe que aceita um tipo em sua declaração pode trabalhar internamente com este tipo. Isso quer dizer que os métodos podem aceitar em seus parâmetros objetos do tipo especificado na criação da classe, retornar esses tipos em propriedades, etc..

Para exemplificarmos, vejamos uma classe que aceita um valor genérico, o que quer dizer, que ela pode trabalhar (fortemente tipada) com qualquer tipo. Através do exemplo abaixo "T" identifica o tipo genérico que, já pode ser acessado internamente pela classe.

```
VB.NET
Public Class ClasseGenerica(Of T)

    Private _valor As T

    Public Property Valor() As T
        Get
            Return Me._valor
        End Get
        Set(ByVal value As T)
            Me._valor = value
        End Set
    End Property

End Class

'Utilização:

Dim classe1 As New ClasseGenerica(Of String)
classe1.Valor = ".NET"
```

```
Dim classe2 As New ClasseGenerica(Of Integer)
classe2.Valor = 123

C#
public class ClasseGenerica<T>
{
    private T _valor;

    public T Valor
    {
        get
        {
            return this._valor;
        }
        set
        {
            this._valor = value;
        }
    }
}

//Utilização:
ClasseGenerica<string> classe1 = new ClasseGenerica<string>();
classe1.Valor = ".NET";

ClasseGenerica<int> classe2 = new ClasseGenerica<int>();
classe2.Valor = 123;
```

O que diferencia uma classe normal de uma classe genérica é o tipo que devemos especificamos durante a criação da mesma. O valor "T" pode ser substituido por qualquer palavra que você achar mais conveniente para a situação e, quando quiser referenciar o tipo genérico em qualquer parte da classe, poderá acessá-lo como um tipo qualquer, como um *Integer* e felizmente, o *Intellisense* dá suporte completo a *Generics*. Digamos que sua classe somente trabalhará com *Streams*, então poderia definir "T" como "TStream" (se assim desejar):

```
VB.NET
Public Class ClasseGenerica(Of TStream)
    '…
End Class

C#
public class ClasseGenerica<TStream>
{
    //…
}
```

Como podemos notar no exemplo, a *classe1* trabalha somente com valores do tipo string. Já a *classe2* somente trabalha com valores do tipo inteiro. Mesmo que quiser adicionar um tipo incompatível, o erro já é informado em *design-time*.

Nota: O Visual C# disponibiliza uma *keyword* bastante útil chamada **default**. Ela é utilizada em classes genéricas para inicializar um tipo qualquer, pois o Visual C# temos obrigatoriamente que definir um valor default para qualquer tipo, esta vem para suprir esta necessidade. Em casos de tipos-referência, é retornado um valor nulo. Já em casos de valores numéricos, 0 é retornado. Quando o tipo é uma estrutura, é retornado para cada membro desta, 0 ou nulo, dependendo do tipo de cada um. O código abaixo exemplifica o uso desta *keyword*:

```
C#
public class Lista<T> where T : IComparable
{
    public void Add(T item)
    {
        T temp = default(T);
        // ....
    }
}
```

Tipos e Termos

Para o entendimento completo de *Generics* é necessário conhecermos alguns termos que são utilizados durante o uso de *Generics*. Inicialmente temos dois termos a serem analisados: *type parameters* e *type arguments*. O primeiro deles refere-se ao parâmetro que é utilizado na criação do tipo genérico. Já os *type arguments* referem-se aos tipos que substituem os *type parameters*. O código abaixo exemplifica essa diferença:

```
VB.NET
Public Class ClasseGenerica(Of T)  'T = type parameter
    '...
End Class

'Utilização
Dim c As New ClasseGenerica(Of String)  'String = type argument

C#
public class ClasseGenerica<T> //T = type parameter
{
    //...
}
```

```
//Utilização
ClasseGenerica<string> c = new ClasseGenerica<string>();
//string = type argument
```

Classes genéricas são também chamadas de *open types*. Levam essa definição porque permitem construirmos uma única classe utilizando diversos tipos. Para ela ser encarada como um *open type* é necessário que a classe tenha, no mínimo, um tipo a ser especificado. Baseando-se no exemplo acima, a classe *ClasseGenerica* trata-se de uma *open type* porque permite, através do *type parameter* "T", especificarmos um tipo que a classe irá manipular. As instâncias de classes *open types* são consideradas *constructed types*.

Além disso, ainda temos mais duas formas de tipos genéricos: *open constructed type* e *close constructed type*. A primeira forma, *open constructed type*, é criado quando, no mínimo, um *type argument* não é especificado, continuando aberto para uma definição futura. Para transformar essa explicação em código, temos o seguinte exemplo:

```
VB.NET
Public Class ClasseGenerica(Of T)
    '...
End Class

Public Class Derivada(Of T)
    Inherits ClasseGenerica(Of T)

End Class

C#
public class ClasseGenerica<T>
{
    //...
}

public class Derivada<T> : ClasseGenerica<T>
{
    //...
}
```

Como podemos notar, o tipo ainda continua aberto para que se possa fazer a definição efetiva do tipo mais tarde. Finalmente, temos o *close constructed type*, que é criado especificando todos os *type arguments*, não permitindo deixá-lo "aberto" para uma futura definição. O exemplo abaixo ilustra o *close constructed type*:


```
VB.NET
```

```
Public Class ClasseGenerica(Of K, T)
    '...
End Class

Public Class Derivada
    Inherits ClasseGenerica(Of String, Cliente)

End Class
```

C#

```
public class ClasseGenerica<K, T>
{
    //...
}

public class Derivada : ClasseGenerica<string, Cliente>
{
    //...
}
```

Mas *Generics* não param por aqui. Existem muitas outras possibilidades para tornarmos os *Generics* ainda mais poderosos, como por exemplo critérios (*constraints*), criação de métodos genéricos, *delegates*, *Interfaces*, entre outros, quais veremos a seguir.

Métodos genéricos

Quando criamos uma classe genérica, informamos o seu tipo logo na construção da mesma. Isso nos habilita a possibilidade de utilizar esse tipo genérico no inteiror da classe em qualquer outro membro. O exemplo abaixo exibe como construir esse metódo:

VB.NET

```
Public Class ClasseGenerica(Of T)
    Public Function RetornaValor(value As T) As T
        Return value
    End Function
End Class
```

C#

```
public class ClasseGenerica<T>
{
    public T RetornaValor(T value)
    {
        return value;
    }
}
```

Como pode notar, quando informamos na construção da classe que ela é uma classe genérica, podemos utilizar o tipo "T" em toda a classe. O método *RetornaValor* recebe esse tipo e o retorna. Quando instanciamos a classe e especificamos o tipo, como por exemplo uma *String*, todos os membros que utilizam "T" passarão a utilizar *String*. Felizmente a verificação de tipos é feito em design-time, o que significa que se tentar passar um número inteiro, ele já acusará o erro, sem a necessidade de executar o código para descobrir a possível falha. Através da imagem 1.4 podemos certificar que ao definir o tipo, a classe e o método somente permitirá trabalhar com ele (onde for definido):

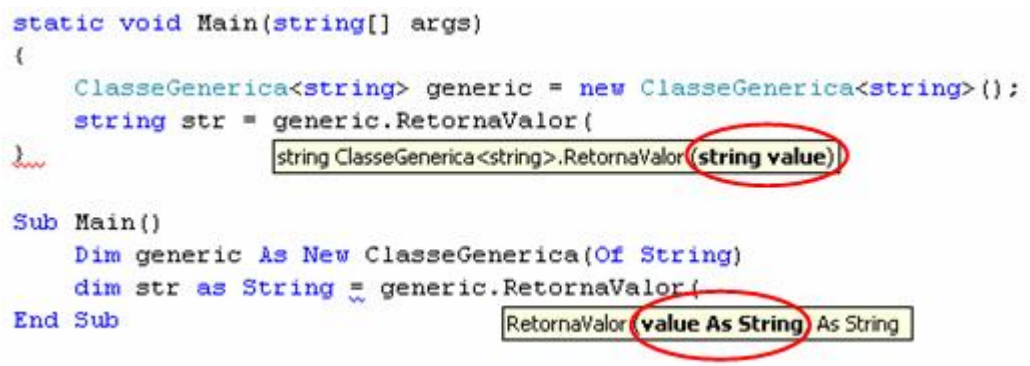

Imagem 1.4 – *Intellisense* dando suporte total a *Generics*.

Se definirmos o tipo da classe como qualquer outro tipo durante a criação do objeto, onde o "T" estiver especificado será substituido por ele. Só que existem situações onde a classe não é genérica, mas sim somente um método de uma classe qualquer deve suportar parâmetros genéricos.

Para isso, há a possibilidade de criarmos métodos genéricos. São basicamente como as classes: definimos o tipo em sua construção e pode-se utilizar tal tipo no interior do respectivo método, seja nos parâmetros ou no retorno do mesmo. Um exemplo da construção de métodos genéricos é exibido abaixo:

VB.NET
```
Public Class Classe
    Public Function RetornaValor(Of T)(ByVal value As T) As T
        Return value
    End Function
End Class
```

C#
```
public class Classe
{
    public T RetornaValor<T>(T value)
    {
        return value;
    }
```

```
}
```

Para criarmos um método genérico que independe de um tipo especificado na criação da classe, devemos colocar logo após o nome do mesmo o tipo genérico que deverá ser especificado pelo desenvolvedor que o está consumindo. Neste caso, o escopo de utilização de "T" (tipo genérico) passa a ser somente o método. Abaixo é mostrado como utilizá-lo:

```
VB.NET
Dim generic As New Classe()

Dim id As Integer = generic.RetornaValor(Of Integer)(12)
Dim nome As String = generic.RetornaValor(Of String)("José")

C#
Classe generic = new Classe();

int id = generic.RetornaValor<int>(12);
string nome = generic.RetornaValor<string>("José");
```

Propriedades genéricas

As propriedades não tem nada de especial. Elas apenas podem retornar um tipo genérico especificado na criação da classe ou *Interface*.

Um exemplo de sua utilização está no primeiro trecho de código desta seção, que é a propriedade *Valor* da classe *ClasseGenerica*. Ela devolve um tipo "T" que é especificado na criação da classe.

Interfaces genéricas

Assim como as classes, as Interfaces também suportam tipos genéricos. A forma de criação também é idêntica a especificação de tipo genérico da classe e, podemos em seu inteiror, utilizar este tipo.

Dentro do .NET Framework 2.0 existem várias *Interfaces* genéricas e estudaremos algumas delas quando estivermos falando sobre coleções genéricas, no Capítulo 2. Já o exemplo de sua criação e implementação é mostrada através do código abaixo:

```
VB.NET
Public Interface ITest(Of T)
    Property Valor() As T
    Sub Adicionar(ByVal value As T)
```

```
End Interface

Public Class ClasseConcreta
    Implements ITest(Of String)

    Public Sub Adicionar(ByVal value As String) _
        Implements ITest(Of String).Adicionar

    End Sub

    Public Property Valor() As String _
        Implements ITest(Of String).Valor
        Get

        End Get
        Set(ByVal value As String)

        End Set
    End Property
End Class
```

C#

```
public interface ITest<T>
{
    T Valor { get;set; }
    void Adicionar(T value);
}

public class ClasseConcreta : ITest<string>
{
    public string Valor
    {
        get
        {
        }
        set
        {
        }
    }

    public void Adicionar(string value)
    {
    }
}
```


Obviamente que quando implementamos a *Interface ITest* em uma classe qualquer, obrigatoriamente devemos definir o tipo que a *Interface* irá trabalhar e, neste caso, o tipo é *String*. Com isso, todos os tipos genéricos especificados na Interface serão implementados utilizando o tipo *String* para esta classe.

Nada impede você de implementar esta mesma *Interface* em uma outra classe com um tipo diferente, ou ainda, implementar a mesma *Interface* com tipos diferentes. A única consideração a fazer quando implementamos a mesma *Interface* em uma classe, pois neste caso, as *Intefaces* são implementadas explicitamente. Veremos a implementação explícitas de *Interfaces* ainda neste capítulo.

Delegates genéricos

Assim como os membros que vimos até o momento, os *delegates* também permitem serem criados de forma genérica. Basicamente levam também o tipo a ser especificado, o que conseqüentemente, somente permitirá apontar para um membro que obrigatoriamente atende aquele tipo.

O exemplo abaixo cria um delegate genérico que aceita um tipo a ser especificado durante a sua criação. Esse delegate poderá ser utilizado para qualquer função que contenha o mesmo número de parâmetros, independentemente do tipo:

```
VB.NET
Public Delegate Sub Del(Of T)(ByVal item As T)

Sub Main()
    Dim delInteger As New Del(Of Integer)(AddressOf WriteInteger)
    Dim delString As New Del(Of String)(AddressOf WriteString)

    delInteger(123)
    delString("José")
End Sub

Public Sub WriteInteger(ByVal i As Integer)
    Console.WriteLine(i)
End Sub

Public Sub WriteString(ByVal str As String)
    Console.WriteLine(str)
End Sub

C#
public delegate void Del<T>(T item);

static void Main(string[] args)
{
    Del<int> delInteger = new Del<int>(WriteInteger);
    Del<string> delString = new Del<string>(WriteString);

    delInteger(123);
    delString("José");
```

```
}

public static void WriteInteger(int i)
{
    Console.WriteLine(i);
}

public static void WriteString(string str)
{
    Console.WriteLine(str);
}
```

O Visual C# ainda tem alguns atalhos para o código acima. O primeiro deles é o chamado de ***method group conversion*** que permite definir o *delegate* genérico sem a necessidade de instanciá-lo. Sendo assim, o código em C# que faz a definição do método a ser executado quando o *delegate* for disparado, poderia ser alterado para:

```
C#
public delegate void Del<T>(T item);

static void Main(string[] args)
{
    Del<int> delInteger = WriteInteger;
    Del<string> delString = WriteString;

    delInteger(123);
    delString("José");
}

//Métodos ocultados
```

Para finalizar, o Visual C# ainda tem um forma de tornar mais simples ainda o processo que são os ***metódos anônimos***. Os métodos anônimos são uma novidade do Visual C# 2.0 e permitem que o desenvolvedor defina o código a ser executado pelo *delegate* diretamente, sem a necessidade de criar um procedimento para isso.

Abaixo temos o código em Visual C# modificado que faz o mesmo que vimos anteriormente, mas agora com o uso dos métodos anônimos:

```
C#
public delegate void Del<T>(T item);

static void Main(string[] args)
{
```

```
    Del<string> delString = new Del<string>(delegate(string str)
    {
        Console.WriteLine(str);
    });

    Del<int> delInteger = new Del<int>(delegate(int i)
    {
        Console.WriteLine(i);
    });

    delInteger(123);
    delString("José");
}
```

Estruturas genéricas

Assim como as classes e *Interfaces*, as estruturas também podem ser genéricas e seguem exatamente as mesmas regras para *Generics*. Um exemplo típico de estrutura genérica é a **Nullable<T>**, discutida na próxima seção, ainda neste capítulo.

Constraints

Vimos até o momento como criar classes, métodos, *Interfaces* e *delegates* genéricos. Eles permitem que o consumidor possa definir qualquer tipo. Mas e se quisermos restringir quais tipos podem ser especificados na criação dos *Generics*? É neste momento que entra em ação as *constraints*. As *constraints* são regras que aplicamos aos tipos genéricos para espeficarmos o que o consumidor pode ou não definir como tipo. Atualmente existem seis tipos de *constraints*:

Constraint	**Descrição**
where T : struct	O tipo especificado deverá ser um value-type, mas precisamente uma estrutura. Mais uma vez, a estrutura **Nullable<T>** é um exemplo.
where T : class	O tipo especificado deverá ser uma reference-type, como classe, *Interface*, *delegate* ou *Array*.
where T : new()	O tipo especificado deverá ter um construtor público sem nenhum parâmetro. Se for utilizado com outras *constraints*, então esta deverá ser a última.
where T : Base Class	O tipo especificado deverá derivar da classe informada.
where T : interface	O tipo especificado deverá implementar a *Interface* informada, podendo inclusive, definir várias *constraints* desse tipo, o que indica que o tipo deverá obrigatoriamente implementar todas elas.
where T : U	Quando há mais de um tipo especificado, podemos definir a *constraint* onde onde obrigatoriamente deverá herdar do outro.

As constraints são bastante úteis, pois apesar de garantir a integridade do seu tipo, ou seja, o consumidor não poderá passar um tipo que possa violar a *constraint*. Além disso, você já é notificado pelo compilador se isso acontecer, que não permite compilar a aplicação com a falha.

Para criar as *constraints*, você utiliza a keyword **where** (no caso do Visual C#). No Visual Basic .NET, a keyword é **As**, listando *constraint* ao lado de *constraint* para a definição do tipo genérico. Abaixo vamos exemplificar como definir cada uma das *constraints* disponíveis:

where T : struct

```
VB.NET
Public Structure Nullable(Of T As Structure)

C#
public struct Nullable<T> where T : struct
```

where T : class

```
VB.NET
Public Class Util(Of T As Class)

C#
public class Util<T> where T : class
```

where T : new()

```
VB.NET
Public Class Util(Of T As New())

C#
public class Util<T> where T : new()
```

where T : Base Class

```
VB.NET
Public Class Util(Of T As BaseReader)

C#
```

```
public class Util<T> where T : BaseReader
```

where T : Interface

```
VB.NET
Public Class Util(Of T As IReader)

C#
public class Util<T> where T : IReader
```

where T : U

```
VB.NET
Public Class Util(Of T, U As T)

C#
public class Util<T, U> where T : U
```

Finalmente, se desejar definir várias *constraints* em um determinado tipo, basta ir enfileirando uma ao lado da outra, como é mostrado logo abaixo. Somente atente-se para o Visual Basic .NET que, neste caso, exige que você envolva as *constraints* entre chaves {}:

```
VB.NET
Public Class Util(Of T As {IComparable, IReader, New})

C#
public class Util<T> where T : IComparable, IReader, new()
```

NullableTypes

Qualquer tipo-valor em .NET possui sempre um valor padrão. Isso quer dizer que ele nunca poderá ter um valor nulo e mesmo que tentar, definindo nulo (*Nothing* em VB.NET e *null* em C#) para o tipo-valor, o compilador atirará uma exceção. Um exemplo é o *DateTime*. Ele tem um valor padrão que é **01/01/0001 00:00:00**. Apesar de estranha, é uma data válida.

Muitas vezes um tipo pode ter uma data não informada, como por exemplo, imagine um objeto *Funcionario* que tem uma propriedade chamada *DataDemissao*. Só que este

funcionario não foi demitido. Então como conseguimos distingüir se essa data é válida ou não? Pois bem, o .NET Framework 2.0 fornece o que chamamos de *Nullable Types*.

System.Nullable<T> é uma estrutura de dados genérica que aceita como tipo qualquer outro tipo, desde que esse tipo seja uma outra estrutura, como por exemplo: *Int32*, *Double*, *DateTime*, etc.. Através deste tipo especial podemos definir valores nulos para ele, como por exemplo:

```
VB.NET
Dim dataDemissao As Nullable(Of DateTime)
dataDemissao = Nothing

C#
Nullable<DateTime> dataDemissao = null;
```

A estrutura genérica **Nullable<T>** fornece também uma propriedade chamada *HasValue* do tipo booleana que retorna um valor indicando se existe ou não um valor definido. E note que a estrutura **Nullable<T>** trata-se de uma estrutura genérica, onde o tipo a ser definido obrigatoriamente deve também ser uma estrutura, devido a *constraint* que obriga isso. Isso pode ser facilmente notado na documentação:

```
VB.NET
Public Structure Nullable(Of T As Structure)

C#
public struct Nullable<T> where T : struct
```

Esses tipos são ideais para utilizá-los junto com registros retornados de uma base de dados qualquer. Apesar de não ter uma grande integração, ajuda imensamente, para conseguirmos mapear as colunas do *result-set* para as propriedades dos objetos da aplicação que permitem valores nulos.

Exceptions

Essa é uma das partes mais elegantes do .NET Framework: Exceções e o tratamento delas. Mas afinal, o que é uma exceção: Exceção é uma violação de alguma suposição da interface do seu tipo. Por exemplo, ao projetar um determinado tipo, você imagina as mais diversas situações em que seu tipo será utilizado, difinindo também seus campos, propriedades, métodos e eventos. Como já sabemos, a maneira como você define esses membros, torna-se a interface do seu tipo.

Assim sendo, dado um método chamado *TransferenciaValores*, que recebe como parâmetro dois objetos do tipo *Conta* e um determinado valor (do tipo *Double*) que deve ser transferido entre elas, precisamos "validá-los" para que a transferência possa ser efetuada com êxito. O desenvolvedor da classe precisará ter conhecimento suficiente para implementar essa tal "validação" e, não esquecer do mais importante: documentar claramente para que os utilizadores deste componente possam implementar o código que fará a chamada ao método da maneira mais eficiente possível, poupando ao máximo que surpresas ocorram em tempo de execução.

VB.NET

```
Public Shared Sub TransferenciaValores(de As Conta, para As
Conta, valor As Double)
    '...
End Sub
```

C#

```
public static void TransferenciaValores(Conta de, Conta para,
double valor){
    //...
}
```

Dentro deste nosso cenário, vamos analisar algumas suposições ("validações") que devemos fazer para que o método acima possa ser executado da forma esperada:

- Certificar que de e para não são nulos
- Certificar que de e para não referenciam a mesma conta
- Se o valor for maior que zero
- Se o valor é maior que o saldo disponível

Necessitamos agora informar ao chamador que alguma dessas regras foi violada e, como fazemos isso? Atirando uma exceção. Como dissemos logo no começo desta seção, ter uma exceção nem sempre é algo negativo na aplicação, pois o tratamento de exceções permite capturar a exceção, tratá-la e a aplicação continuará correndo normalmente.

O tratamento delas dá se de forma estrutura, utilizando o *Try/Catch/Finally*. Mas nem sempre é necessário ter o overhead desta estrutura, pois há algumas formas de assegurar que o código não dará erros com outros métodos, também fornecidos pelo .NET Framework. Para exemplificar, vamos comparar dois códigos: o primeiro, apesar de resolver o problema, não é a melhor forma; já o segundo pode ser a melhor alternativa para a resolução, ou melhor, para evitar um possível problema:

C#

```
//Código ruim

try
{
    IReader reader = (IReader)e.Data;
    reader.ExecuteOperation();
}
catch
{
    Console.WriteLine("Tipo incompatível.");
}

try
{
    string tempId = "12";
    int id = Convert.ToInt32(tempId);
    this.BindForm(id);
}
catch
{
    Console.WriteLine("Id inválido.");
}

//Código reformulado

IReader reader = (IReader)e.Data as IReader;
if(reader != null)
{
    reader.ExecuteOperation();
}
else
{
    Console.WriteLine("Tipo incompatível.");
}

string tempId = "12";
int id = 0;
if(int.TryParse(tempId, out id))
{
    this.BindForm(id);
}
else
{
    Console.WriteLine("Id inválido.");
}
```


VB.NET

```
'Código ruim

Try
    Dim reader As IReader = CType(e.Data, IReader)
    reader.ExecuteOperation()
Catch
    Console.WriteLine("Tipo incompatível.")
End Try

Try
    Dim tempId As String = "12"
    Dim id As Integer = Convert.ToInt32(tempId)
    Me.BindForm(id)
Catch
    Console.WriteLine("Id inválido.")
End Try

'Código reformulado

Dim reader As IReader = TryCast(e.Data, IReader)
If Not IsNothing(reader) Then
    reader.ExecuteOperation()
Else
    Response.Write("Tipo incompatível.")
End if

Dim tempId As String = "12"
Dim id As Integer = 0
If Integer.TryParse(tempId, id) Then
    Me.BindForm(id)
Else
    Console.WriteLine("Id inválido.")
End if
```

Veremos mais tarde, ainda neste capítulo, sobre os operadores de conversão.

Atributos

Os atributos são tags declarativas quais podemos decorar nossos membros (assemblies, classes, métodos, propriedades, etc.) de acordo com a necessidade e com a característica do atributo. Esses atributos são armazenados junto aos metadados do elemento e fornecem informações para o runtime que o utilizará para extrair informações em relação ao membro em que é aplicado.

Dentro do .NET Framework existem uma porção de atributos, como é o caso do **CLSCompliant** que vimos no início deste capítulo e também do atributo **WebMethod** que é utilizado para expor um método via *WebService*. Há a possibilidade de

configurarmos inteiramente a segurança de um componente através da forma declarativa, que é justamente utilizando atributos.

Para que você possa criar o teu próprio atributo, você obrigatoriamente deve derivar da classe base chamada **System.Attribute**. Imagine que você tenha uma classe chamada **Cliente**. Essa classe possui várias propriedades e, entre elas, a propriedade **Nome**. Você cria uma coleção de clientes e a popula com todos os clientes da sua empresa.

Em um segundo momento, você cria um atributo chamado **ReportAttribute** que pode ser aplicado exclusivamente em propriedades. Esse atributo indica quais propriedades devem fazer parte de um relatório qualquer. Para exemplificar apenas o uso do atributo em conjunto com a propriedade, analise o código a seguir:

VB.NET

```
<AttributeUsage(AttributeTargets.Property)> _
Public Class ReportAttribute
    Inherits Attribute
End Class

Public Class Cliente

    Private _nome As String

    <Report()> _
    Public Property Nome() As String
        Get
            Return Me._nome
        End Get
        Set(ByVal value As String)
            Me._nome = value
        End Set
    End Property

    ' outras propriedades

End Class
```

C#

```
[AttributeUsage(AttributeTargets.Property)]
public class ReportAttribute : Attribute
{
}

public class Cliente
{
    private string _nome;

    [Report]
```

```
    public string Nome
    {
        get
        {
            return this._nome;
        }
        set
        {
            this._nome = value;
        }
    }

    // outras propriedades
}
```

Como podemos reparar, na criação da classe **ReportAttribute** herdamos diretamente da classe **Attribute**. Denotamos esse atributo customizado com um outro atributo fornecido pelo .NET Framework que é o **AttributeUsage** que permite especificamos em qual tipo de membro podemos aplicar o atributo. Para o nosso cenário, vamos pode aplicá-lo somente em propriedades.

Depois dele pronto, basta simplesmente aplicar o mesmo nas propriedades que qualquer objetos customizado. Claro que precisaríamos ainda de uma função que extrai esses atributos via reflexão (**System.Reflection**) para assim tomarmos uma decisão baseando-se no atributo. Vale lembrar também que é perfeitamente possível termos propriedades nos atributos customizados para assim passarmos mais informações extras a nível de metadados ou até mesmo informações de regras de negócios. Para finalizar, qualquer membro pode suportar um ou vários atributos.

Trabalhando com Interfaces

O que são Interfaces?

Uma *Interface* se assemelha a uma classe. Ela pode conter métodos, propriedades e eventos, mas sem qualquer implementação de código, ou seja, somente contém a assinatura dos membros. As *Interfaces* são implementadas em classes e estas, por sua vez, implementam todo o código para os membros abstratos da *Interface*, colocando ali o código específico para a classe que a implementa.

As *Interfaces* são úteis para arquiteturas de softwares que exigem um ambiente *plug-and-play*. Isso quer dizer que uma *Interface* pode referenciar armazenar uma instância de uma classe concreta desde que esta instância seja um tipo da *Interface*. Enteda-se por "ser um tipo de" uma classe que implementa a *Interface* em questão.

As *Interfaces* podem ser implementadas em várias e classes e uma classe pode implementar várias *Interfaces*. Quando uma classe implementa uma Interface qualquer,

isso assegurará que o classe fornece exatamente os mesmos membros especificados na *Interface* implementada de forma pública. A *Interface* é uma forma de "contrato", já que deixa explícito exatamente os membros que a classe deverá implementar.

O .NET Framework usa *Interfaces* por toda sua arquitetura e nos permite criar nossas próprias *Interfaces*. Além disso, o .NET Framework também expõe *Interfaces* que nos fornecem o "contrato" para implementarmos em nossos objetos para garantir que eles implementem exatamente os membros que são necessários para outros tipos poderem manuseá-lo. Entre as inúmeros *Interfaces* fornecidas, vamos analisar algumas delas, as mais úteis ao nosso dia-à-dia e entender como e quando implementá-las.

Quando falamos de implementação de Interfaces, há dois tipos:

1. **Normal:** Esta opção é a mais tradicional, pois implementamos a *Interface* em uma classe qualquer e os métodos, propriedades e eventos definidos pela *Interface* são publicamente disponibilizados também pela classe.
2. **Explícita:** A opção implícita permite-nos implementar várias *Interfaces* com membros com o mesmo nome em um determinado objeto. Isso é um caso comum, pois há situações onde a *Interface* tem um método comum com outra *Interface* mas ambas precisam ser implementadas.

A implementação explícita são suportadas tanto no Visual Basic .NET quanto no Visual C#, mas há uma pequena diferença: o Visual C# prefixa o nome do método com o nome da *Interface* correspondente; já o Visual Basic .NET cria nomes diferentes para o método, mas permite invocá-lo através da *Interface*. Além disso, o Visual Basic .NET ainda permite que se invoque os métodos através da instância da classe o que não é permitido no Visual C#. Abaixo é exibido um exemplo deste cenário:

```
VB.NET
Public Interface IReader
    Sub Start()
End Interface

Public Interface IWriter
    Sub Start()
End Interface

Public Class Process
    Implements IReader
    Implements IWriter

    Public Sub Start() Implements IReader.Start
        Console.WriteLine("IReader.Start")
    End Sub

    Public Sub Start1() Implements IWriter.Start
```

```
        Console.WriteLine("IWriter.Start")
    End Sub
End Class
```

C#

```
public interface IReader {
    void Start();
}

public interface IWriter {
    void Start();
}

public class Process : IReader, IWriter
{
    void IReader.Start()
    {
        Console.WriteLine("IReader.Start");
    }

    void IWriter.Start()
    {
        Console.WriteLine("IWriter.Start");
    }
}
```

Como podemos ver, as *Interfaces IReader* e *IWriter* possui um mesmo método chamado *Start*. A única observação é que no caso do Visual Basic .NET ele necessariamente precisa ter um nome diferente na classe concreta *Process*. No caso do Visual C#, o prefixo do método, que é o nome da *Interface*, garante a distinção entre elas.

As diferenças não param aí. No caso do Visual C# esses métodos somente são acessíveis se forem invocados através da própria *Interface*. Isso quer dizer que, através da instância da classe *Process* não está disponível e, se reparar, o *Intellisense* também não suporta. Isso já é possível no Visual Basic .NET. O código abaixo mostra a utilização da classe *Process*:

VB.NET

```
Dim p As New Process
Dim reader As IReader = p
Dim writer As IWriter = p
reader.Start()
writer.Start()

'Mas também é possível fazer:
p.Start()
p.Start1()
```

```
C#
Process p = new Process();
IReader reader = (IReader)p;
IWriter writer = (IWriter)p;
reader.Start();
writer.Start();
```

Esse tipo de implementação somente se faz necessária quando há necessidade de implementar duas *Interfaces* em um objeto que contém exatamente um mesmo membro. Um exemplo disso dentro do .NET Framework são as estruturas de dados, como por exemplo: *Int32*, *Double*, etc., elas implementam explicitamente a *Interface* **IConvertible**.

Conversão de tipos utilizando IConvertible

Imagine que esteja desenvolvendo uma aplicação em que contém uma classe chamada *CalculoDeMedicoesPredio* e dentro dela toda a lógica para calcular as medições de uma construção qualquer. Agora, você precisa deste valor para gerar outras informações para o usuário, como por exemplo relatórios ou até mesmo gráficos.

Geralmente, os módulos que esperam esses valores (que talvez não seja nem a sua aplicação, pode ser um componente de terceiros) exigem que os valores estejam em um determinado tipo. Sendo assim, como eu faço para converter meu tipo complexo (*CalculoDeMedicoesPredio*) em um tipo esperado pelo módulo para geração de relatórios e gráficos?

O .NET Framework fornece uma *Interface* chamada **IConvertible** que permite-nos implementá-la em nossos objetos. Essa *Interface* possuiu uma série de métodos que permitem converter um objeto em um tipo fornecedido pelo .NET Framework, a saber: *Boolean*, *SByte*, *Byte*, *Int16*, *UInt16*, *Int32*, *UInt32*, *Int64*, *UInt64*, *Single*, *Double*, *Decimal*, *DateTime*, *Char*, e *String*. Os nomes dos métodos da *Interface* em questão são basicamente os mesmos, apenas prefixados com **To**. Exemplo: *ToBoolean*, *ToDateTime*, *ToString*, etc.. Como já devem desconficar, implementando a *Interface IConvertible* em seu tipo, poderá utilizá-lo passando para o método correspondente a conversão que quer fazer para a classe **Convert**.

Um ponto importante aqui é que você precisa antentar-se é para quais tipos o seu objeto pode ser convertido. No caso acima, não faz sentindo convertermos a classe *CalculoDeMedicoesPredio* para *Boolean*. Nos tipos "incompatíveis" o ideal é sempre você atirar uma exceção do tipo **InvalidCastException** para não permitem conversões não suportadas.

Através do código abaixo é possível analisar como implementar a *Interace IConvertible*:

VB.NET

```
Public Class CalculoDeMedicoesPredio
    Implements IConvertible

    Private _altura As Double
    Private _largura As Double

    Public Sub New(ByVal largura As Double, _
        ByVal altura As Double)

        Me._altura = altura
        Me._largura = largura
    End Sub

    Public Function ToBoolean(ByVal provider As IFormatProvider)
As Boolean Implements IConvertible.ToBoolean

        Throw New InvalidCastException
    End Function

    Public Function ToDouble(ByVal provider As IFormatProvider)
As Double Implements IConvertible.ToDouble

        Return Me._altura * Me._largura
    End Function

    'demais métodos
End Class

'Utilização:

Dim calculo As New CalculoDeMedicoesPredio(10, 30)
Dim area As Double = Convert.ToDouble(calculo)
Console.WriteLine(area)
```

C#

```
public class CalculoDeMedicoesPredio : IConvertible
{
    private double _largura;
    private double _altura;

    public CalculoDeMedicoesPredio(double largura, double altura)
    {
        this._altura = altura;
        this._largura = largura;
    }

    public bool ToBoolean(IFormatProvider provider)
    {
        throw new InvalidCastException;
```

```
    }

    public double ToDouble(IFormatProvider provider)
    {
        return this._altura * this._largura;
    }

    // demais métodos
}

//Utilização:

CalculoDeMedicoesPredio calculo =
    new CalculoDeMedicoesPredio(10, 30);
double area = Convert.ToDouble(calculo);
Console.WriteLine(area);
```

IComparer, IComparable e IEquatable

Geralmente quando estamos trabalhando com objetos customizados dentro da aplicação, é muito normal querermos exibir tais objetos para o usuário. Para exemplificar, teremos um objeto do tipo *Usuario* que possui uma propriedade do tipo *Nome*.

Imagine que você tenha uma lista de usuários com nomes aleatórios, ou seja, sem uma ordenação específica. Essa lista conterá objetos do tipo *Usuario* com o seu respectivo nome. A aplicação se encarrega de carregar essa lista e chega o nomento em que você precisa exibí-la em um controle qualquer e, é necessário que essa lista seja exibida aplicando uma ordenação alfabética na mesma. Como proceder para customizar a ordenação?

Pois bem, várias coleções no .NET Framework, como por exemplo *List<T>*, *ArrayList*, etc., fornecem um método chamado *Sort*. Só que apenas invocar este método para ordenar a lista não é o suficiente. É necessário que você implemente as *Interfaces* **IComparer<T> (IComparer)** e **IComparable<T> (IComparable)** para isso. A primeira delas, **IComparer<T>**, customiza a forma de como ordenar a coleção; já a segunda, **IComparable<T>**, é utilizada como definir como a ordenação é realizada em uma classe específica.

Inicialmente iremos analisar a *Interface* **IComparable<T>**. Ela fornece um método chamado *CompareTo*, que permite comparar a instância da uma classe onde esta *Interface* está sendo implementando com um objeto (do mesmo tipo) que é passado para o método. Com a versão 2.0 do .NET Framework foi criada uma versão nova desta mesma *Interface* **IComparable**, que é a **IComparable<T>**. A diferença em relação a **IComparable** é que permite trabalharmos com *Generics*, o que melhora muito a performance, já que não é necessário efetuarmos a conversão de **System.Object** para a classe que estamos utilizando e, somente depois disso, compararmos.

Com todos os conceitos definidos, resta-nos partir para a implementação concreta para analisarmos:

VB.NET

```
Public Class Usuario
    Implements IComparable(Of Usuario)

    Private _nome As String

    Public Sub New(ByVal nome As String)
        Me._nome = nome
    End Sub

    Public Property Nome() As String
        Get
            Return Me._nome
        End Get
        Set(ByVal value As String)
            Me._nome = value
        End Set
    End Property

    Public Function CompareTo(ByVal other As Usuario) As _
        Integer Implements IComparable(Of Usuario).CompareTo

        Return Me.Nome.CompareTo(other.Nome)
    End Function
End Class
```

C#

```
public class Usuario : IComparable<Usuario>
{
    private string _nome;

    public Usuario(string nome)
    {
        this._nome = nome;
    }

    public string Nome
    {
        get { return _nome; }
        set { _nome = value; }
    }

    public int CompareTo(Usuario other)
    {
        return this.Nome.CompareTo(other.Nome);
```

```
    }
}
```

Ao especificar a implementação da *Interface* **IComparable<T>**, já definimos que esta *Interface* deverá trabalhar com o tipo *Usuario*, que é justamente a classe onde estamos implementando-a. Dentro do método *CompareTo* fornecido por ela comparamos a propriedade *Nome* da instância corrente com a propriedade *Nome* da instância que vem como parâmetro para método. Reparem que utilizando a *Interface* genérica não é necessário efetuar a conversão dentro do método para, em seguinda, efetuar a comparação, pois o objeto já está com o tipo especificado na implementação da *Interface*. Ao exibir a coleção depois de invocar o método *Sort*, o resultado será o seguinte:

VB.NET

```
Dim list As New List(Of Usuario)
list.Add(New Usuario("Virginia"))
list.Add(New Usuario("Zuleika"))
list.Add(New Usuario("Ana"))
list.Add(New Usuario("Elisabeth"))
list.Add(New Usuario("Tiago"))
list.Add(New Usuario("Humberto"))

list.Sort()

'Output
Ana
Elisabeth
Humberto
Tiago
Virginia
Zuleika
```

C#

```
List<Usuario> list = new List<Usuario>();
list.Add(new Usuario("Virginia"));
list.Add(new Usuario("Zuleika"));
list.Add(new Usuario("Ana"));
list.Add(new Usuario("Elisabeth"));
list.Add(new Usuario("Tiago"));
list.Add(new Usuario("Humberto"));

list.Sort();

//Output
Ana
Elisabeth
Humberto
Tiago
```

```
Virginia
Zuleika
```

Um ponto bastante importante que deve ser comentado com relação ao código acima é que, ao invocar o método *Sort* e o objeto especificado não implementar a *Interface* **IComparable**, uma exceção do tipo **InvalidOperationException** será lançada.

Como o exemplo é um tanto quanto simples, pois tem apenas uma propriedade e, conseqüentemente, um único campo de ordenação, como ficaria se quiséssemos disponibilizar várias formas para ordenação? Exemplo: inicialmente a listagem é exibida com os nomes em ordem alfabética. Em seguida, gostaria de exibir a mesma listagem, mas agora ordenando por grupo. Para isso, devemos utilizar a *Interface* **IComparer** ou **IComparer<T>**. Essa *Interface* fornece um método denominado *Compare*, que compara dois objetos retornando um número inteiro indicando se um objeto é menor, igual ou maior que o outro objeto. Esta *Interface* deverá ser implementada em uma classe diferente da atual (*Usuario*), uma espécie de classe de "utilidades" para executar o seu trabalho.

O primeiro passo será a criação da classe que implementará a *Interface* **IComparer<T>** e, sem seguida, codificar o método *Compare*. Devemos ainda criar um enumerador para que o consumidor da classe possa escolher por qual campo ele deseja ordenar a listagem. A implementação é exibida abaixo:

VB.NET

```
Public Class UsuarioSorting
    Implements IComparer(Of Usuario)

    Public Enum SortType
        Nome
        Grupo
    End Enum

    Private _sortType As SortType

    Public Sub New(ByVal sortType As SortType)
        Me._sortType = sortType
    End Sub

    Public Function Compare(ByVal x As Usuario, ByVal y As
Usuario) _
        As Integer Implements IComparer(Of Usuario).Compare

        Select Case Me._sortType
            Case SortType.Grupo
                Return x.Grupo.CompareTo(y.Grupo)
            Case SortType.Nome
```

```
                Return x.Nome.CompareTo(y.Nome)
        End Select

        Return 0
    End Function
End Class
```

C#

```
public class UsuarioSorting : IComparer<Usuario>
{
    public enum SortType
    {
        Nome,
        Grupo
    }

    private SortType _sortType;

    public UsuarioSorting(SortType sortType)
    {
        this._sortType = sortType;
    }

    public int Compare(Usuario x, Usuario y)
    {
        switch (this._sortType)
        {
            case SortType.Grupo:
                return x.Grupo.CompareTo(y.Grupo);
            case SortType.Nome:
                return x.Nome.CompareTo(y.Nome);
        }

        return 0;
    }
}
```

A classe UsuarioSorting é responsável por receber qual o tipo de ordenação e, baseado nele irá determinar qual das propriedades devem ser comparadas para montar a ordenação. Finalmente, a utilizadação desta classe muda ligeiramente ao que vimos anteriormente durante o exemplo da *Interface* **IComparable<T>**. A diferença resume a informar para um dos *overloads* do método *Sort* da coleção um objeto que customiza o ordenação (que obrigatoriamente deve implementar a *Interface* **IComparer<T>**):

VB.NET

```
'ordenar por nome
usuarios.Sort(New UsuarioSorting(UsuarioSorting.SortType.Nome))
```

```
'ordenar por grupo
usuarios.Sort(New UsuarioSorting(UsuarioSorting.SortType.Grupo))

C#
//ordenar por nome
usuarios.Sort(new UsuarioSorting(UsuarioSorting.SortType.Nome));

//ordenar por grupo
usuarios.Sort(new UsuarioSorting(UsuarioSorting.SortType.Grupo));
```

Nota: Quando ambas *Interfaces* (**IComparable<T>** e **IComparer<T>** são implementadas, somente a *Interface* **IComparer<T>** é utilizada.

Ainda temos a *Interface* **IEquatable<T>** que compara dois objetos quanto à igualdade, fazendo basicamente o que já faz o método *Equals* de **System.Object**, mas assim como o **IComparable<T>**, com uma vantagem: trabalha com o tipo específico e não **System.Object**, o que evita o *boxing/unboxing*.

Essa *Interface* tem um método chamado *Equals* que recebe como parâmetro um objeto do mesmo tipo especificado na declaração da *Interface*. A implementação da *Interface* é exibida abaixo:

```
VB.NET
Public Class Usuario
    Implements IEquatable(Of Usuario)

    Private _nome As String

    Public Property Nome() As String
        Get
            Return Me._nome
        End Get
        Set(ByVal value As String)
            Me._nome = value
        End Set
    End Property

    Public Function Equals1(ByVal other As Usuario) As _
        Boolean Implements IEquatable(Of Usuario).Equals

        If IsNothing(other) Then Return False
        Return Me.Nome = other.Nome
    End Function
End Class

C#
```

```
public class Usuario : IEquatable<Usuario>
{
    private string _nome;

    public Usuario(string nome)
    {
        this._nome = nome;
    }

    public string Nome
    {
        get { return _nome; }
        set { _nome = value; }
    }

    public bool Equals(Usuario other)
    {
        if(other == null) return false;
        return other.Nome == this.Nome;
    }
}
```

É natural que ao utilizar a classe e chamar o método *Equals*, verá que na lista de *overloads* aparecerá duas versões do mesmo método. Uma delas espera exatamente o tipo informado na *Interface* **IEquatable<T>**, enquanto a outra espera um **System.Object**, que obviamente herda do também de **System.Object**.

Copiando a referência de um objeto utilizando ICloneable

Quando falamos a respeito de tipos-referência citamos que quando atribuímos uma instância de um objeto qualquer a um objeto do mesmo tipo recém criado, ambas passam a apontar (ter a mesma referência) ao mesmo objeto na memória *Heap*. Sendo assim, qualquer mudança efetuada em algum dos objetos, será imediatamente refletida no outro.

Mas pode existir situações onde será necessário criarmos uma cópia, ou melhor, um clone de objeto para que essa operação não interfira em seu original. Para assegurar isso, o Microsoft disponibilizou uma *Interface* chamada **ICloneable** que cria uma nova instância do objeto em questão com os mesmo valores da instância corrente. Esse processo é chamado de "clonar" o objeto e, como já era de se esperar, a Interface fornece um método chamado *Clone* para efetuarmos essa operação. Mas, quando falamos em clone, existem dois tipos:

1. **Shallow-cloning:** Este tipo de clone se resume a copiar apenas os valores, sem copiar qualquer referência a outros objetos que existam internamente ao objeto.Com isso, tipos-referência continuaram apontando para o mesmo local, mesmo no clone.

2. **Deep-cloning:** Este tipo de clone, além de copiar os valores, também copia os membros que são tipo-referências, criando assim um novo objeto completamente independente.

Através das imagens abaixo conseguimos comparar as diferências entre os tipos de clones suportados:

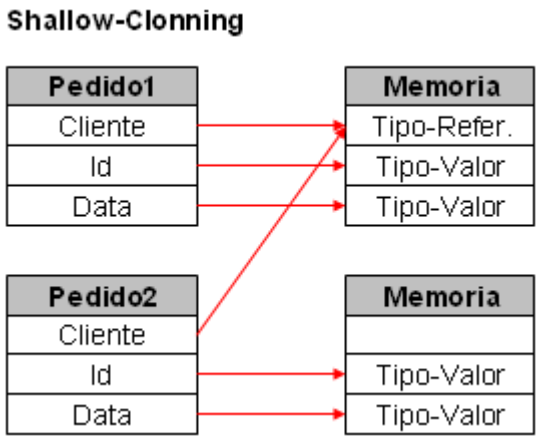

Imagem 1.5 – *Shallow-Clone* – Os tipos-referência ainda continuam apontando para o mesmo local.

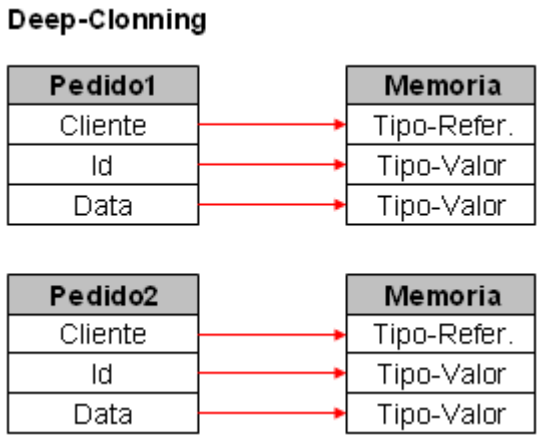

Imagem 1.6 – *Deep-Clone* – Uma cópia completa do objeto.

A classe **System.Object** fornece um método protegido (*protected*) chamado *MemberwiseClone*. Este método retorna um clone no estilo Shallow, ou seja, copiará os valores e apenas as referência para os membros tipo-referência, se existirem. Dependendo do cenário isso não é muito interessante.

Se desejar mesmo fazer um clone do tipo *Deep-Clonning*, será necessário implementar a *Interface* **ICloneable** e customizar o método *Clone*. A implementação não é complexa e podemos comprovar através do código abaixo:

VB.NET

```
Public Class Cliente

    Private _nome As String

    Public Sub New(ByVal nome As String)
        Me._nome = nome
    End Sub

    Public Property Nome() As String
        Get
            Return Me._nome
        End Get
        Set(ByVal value As String)
            Me._nome = value
        End Set
    End Property

End Class

Public Class Pedido
    Implements ICloneable

    Private _id As Integer
    Private _data As DateTime
    Private _cliente As Cliente

    Public Sub New(ByVal id As Integer, ByVal data As DateTime)
        Me._data = data
        Me._id = id
    End Sub

    Public Property Cliente() As Cliente
        Get
            Return Me._cliente
        End Get
        Set(ByVal value As Cliente)
            Me._cliente = value
        End Set
    End Property


    Public Function Clone() As Object _
        Implements System.ICloneable.Clone

        Dim pedidoClone As New Pedido(Me._id, Me._data)
        pedidoClone.Cliente = New Cliente(Me.Cliente.Nome)
        Return pedidoClone
    End Function
```

```
End Class

'Utilização:

Dim pedido1 As New Pedido(1, DateTime.Now)
pedido1.Cliente = New Cliente("José Torres")

Dim pedido2 As Pedido = DirectCast(pedido1.Clone(), Pedido)
pedido2.Cliente.Nome = "Maria Torres"

Console.WriteLine(pedido1.Cliente.Nome)
Console.WriteLine(pedido2.Cliente.Nome)

'Output:
'José Torres
'Maria Torres
```

C#

```
public class Cliente
{
    private string _nome;

    public Cliente(string nome)
    {
        this._nome = nome;
    }

    public string Nome
    {
        get
        {
            return this._nome;
        }
        set
        {
            this._nome = value;
        }
    }
}

public class Pedido : ICloneable
{
    private int _id;
    private DateTime _data;
    private Cliente _cliente;

    public Pedido(int id, DateTime data)
    {
        this._data = data;
        this._id = id;
```

```
    }

    public Cliente Cliente
    {
        get
        {
            return this._cliente;
        }
        set
        {
            this._cliente = value;
        }
    }

    public object Clone()
    {
        Pedido clone = new Pedido(this._id, this._data);
        clone.Cliente = new Cliente(this.Cliente.Nome);
        return clone;
    }
}

//Utilização:

Pedido pedido1 = new Pedido(1, DateTime.Now);
pedido1.Cliente = new Cliente("José Torres");

Pedido pedido2 = (Pedido)pedido1.Clone();
pedido2.Cliente.Nome = "Maria Torres";

Console.WriteLine(pedido1.Cliente.Nome);
Console.WriteLine(pedido2.Cliente.Nome);

//Output:
//José Torres
//Maria Torres
```

Como podemos notar, o clone copiou na íntegra todos os valores do *pedido1*, mesmo os valores que são tipos-referência. Agora, se apenas modificarmos a implementação do método *Clone* retornando a chamada para o método *MemberwiseClone* de **System.Object**, veremos que apenas a referência para o objeto *Cliente* é copiado e, sendo assim, ao alterar a propriedade *Nome* do *pedido2* refletirá também no *pedido1*.

Formatando um tipo em uma string com IFormattable

Já vimos e utilizamos várias vezes padrões de formatação que são fornecidos intrinsicamente pela infraestrutura do .NET Framework desde a sua versão 1.x. Esses padrões são comuns quando necessitamos formatar datas ou números, para exibirmos ao

usuário um valor mais amigável e, alguns exemplos típicos estão neste artigo. Mas há situações em que precisamos formatar o nosso objeto customizado e, felizmente, o .NET Framework fornece uma interface chamada IFormattable qual possui um único método denominado, *ToString* qual é invocado automaticamente pelo runtime quando especificamos uma formatação.

Para o nosso cenário de exemplo, teremos dois objetos: *Cliente* e *Cnpj*. Cada cliente obrigatoriamente terá uma propriedade do tipo *Cnpj*. Este tipo por sua vez, implementará a interface **IFormattable** e deverá fornecer dois tipos de formatação: ***DF*** e ***PR***. O primeiro significa "*Documento Formatado*" e retornará o valor do *CNPJ* formatado; já a segunda opção significa "*Prefixo*" e, se o usuário optar por este tipo de formatação, será retornado apenas o prefixo do *CNPJ* que, para quem não sabe, trata-se dos 9 primeiros dígitos. A arquitetura de classes que servirá como exemplo:

```
VB.NET
Public Class Cliente
    Private _nome As String
    Private _cnpj As Cnpj

    Public Sub New(nome As String, cnpj As String)
        Me._nome = nome
        Me._cnpj = New Cnpj(cnpj)
    End Sub

    Public Property Nome() As String
        Get
            Return Me._nome
        End Get
        Set(Value As String)
            Me._nome = value
        End Set
    End Property

    Public Property Cnpj() As Cnpj
        Get
            Return Me._cnpj
        End Get
        Set(Value As Cnpj)
            Me._cnpj = value
        End Set
    End Property
End Class

Public Class Cnpj
    Implements IFormattable

    Private _numero As String
```

```
    Public Sub New(numero As String)
        Me._numero = numero
    End Sub

    Public Overloads Function ToString(format As String, _
        formatProvider As IFormatProvider) As String _
        Implements IFormattable.ToString

        If format = "DF" Then
            Return
Convert.ToDouble(Me._numero).ToString("000\.000\.000\/0000\-00")
        Else If format = "PR" Then
            Return       Convert.ToDouble(Me._numero.Substring(0,
9)).ToString("000\.000\.000")
        End If

        Return Me._numero
    End Function

    Public Overrides Function ToString() As String
        Return Me.ToString(Nothing, Nothing)
    End Function
End Class

'Utilização:

Dim c As New Cnpj("999999999999999")
Response.Write(string.Format("O  documento  formatado  é  {0:DF}.",
c))
Response.Write(string.Format("O prefixo é {0:PR}.", c))
Response.Write(string.Format("O documento é {0}.", c))

' Output:

' O documento formatado é 999.999.999/9999-99.
' O prefixo é 999.999.999.
' O documento é 999999999999999.
```

C#

```
public class Cliente
{
    private string _nome;
    private Cnpj _cnpj;

    public Cliente(string nome, string cnpj)
    {
        this._nome = nome;
        this._cnpj = new Cnpj(cnpj);
    }
```

```
    public string Nome
    {
        get
        {
            return this._nome;
        }
        set
        {
            this._nome = value;
        }
    }

    public Cnpj Cnpj
    {
        get
        {
            return this._cnpj;
        }
        set
        {
            this._cnpj = value;
        }
    }
}

public class Cnpj : IFormattable
{
    private string _numero;

    public Cnpj(string numero)
    {
        this._numero = numero;
    }

    public   string   ToString(string   format,   IFormatProvider
formatProvider)
    {
        if (format == "DF")
            return

Convert.ToDouble(this._numero).ToString(@"000\.000\.000\/0000\-
00");
        else if (format == "PR")
            return
                Convert.ToDouble(this._numero.Substring(0,
9)).ToString(@"000\.000\.000");

        return this._numero;
    }
```

```
    public override string ToString()
    {
        return this.ToString(null, null);
    }
}


//Utilização:

Cnpj c = new Cnpj("999999999999999");
Response.Write(string.Format("O documento formatado é {0:DF}.",
c));
Response.Write(string.Format("O prefixo é {0:PR}.", c));
Response.Write(string.Format("O documento é {0}.", c));

// Output:

// O documento formatado é 999.999.999/9999-99.
// O prefixo é 999.999.999.
// O documento é 999999999999999.
```

Como podemos reparar, a classe *Cliente* tem uma propriedade chamada *Cnpj* do tipo *Cnpj*. O objeto *Cnpj* implementa a *Interface* **IFormattable** e, dentro do método *ToString*, verifica qual o formato está sendo passado para ele. Baseando-se neste formato é que uma determinada formatação é aplicada ao número do *CNPJ* e, caso nenhuma formatação é especificada, somente o número é retornado, sem nenhuma espécie de formatação.

Esse tipo de formatação torna tudo muito flexível e, podemos ainda especificar o tipo de formatação em controles *DataBound*, como por exemplo o **GridView** do ASP.NET, da mesma forma que fazemos para datas e números. As imagens abaixo ilustram isso:

Nome	CNPJ
Cliente 1	111.111.111/1111-11
Cliente 2	222.222.222/2222-22
Cliente 3	333.333.333/3333-33
Cliente 4	444.444.444/4444-44

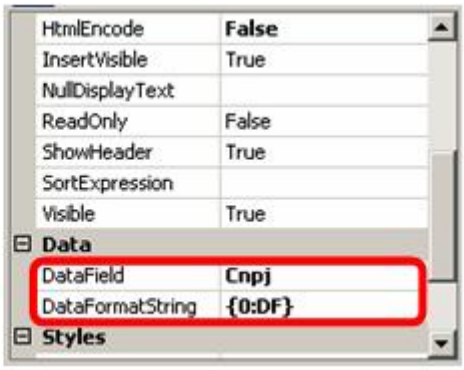

Imagem 1.7 – Aplicando formatação em controles.

Liberando referências a objetos não gerenciados através da interface IDisposable

Utilizar o método *Finalize* disponibilizado pela classe **System.Object** é muito útil, pois assegura que recursos não gerenciados não fiquem "perdidos" quando objetos gerenciados é descartado. Só que há um problema neste cenário: não é possível saber quando o método *Finalize* é chamado e o fato deste método não ser público, não há a possibilidade de invocá-lo explicitamente.

Imagine o seguinte cenário: a conexão com um banco de dados qualquer é extremamente custosa e deixar ela aberta até que o runtime determine que ela seja descartada, podemos ter vários danos, pois isso pode acontecer até alguns dias depois. Para termos a possibilidade de finalizar o objeto deterministicamente e, ali fecharmos conexões com bancos de dados e arquivos a Microsoft implementou um padrão chamado **Dispose**.

Através de uma *Interface* chamada **IDisposable**, que contém um único método chamado ***Dispose***, você pode implementar na sua classe de acesso a dados, arquivos, MessageQueue, entre outros recursos para que o recurso seja explicitamente fechado quando o processo finalizar, sem a necessidade de aguardar que o *Garbage Collector* decida fazer isso.

Na maioria dos cenário onde se utilizar o padrão *Dispose*, dentro da implementação do método invoca-se um outro método chamado *SuppressFinalize* da classe *GC* (*Garbage Collector*). Teoricamente como todos os recursos são finalizados dentro do método *Dispose*, não há necessidade do *runtime* invocar o método *Finalize*, justamente porque não tem mais nada a fazer, pois o objeto já foi descartado. Abaixo podemos visualizar uma classe customizada que implementa o padrão *Dispose*:

```
VB.NET
Public Class Reader

    Implements IDisposable

    Private _reader As StreamReader
    Private disposed As Boolean

    Public Sub New(ByVal filename As String)
        Me._reader = New StreamReader(filename)
    End Sub

    Protected Overridable Sub Dispose(ByVal disposing As Boolean)
        If Not Me.disposed Then
            If disposing Then
                If Not IsNothing(Me._reader) Then
                    Me._reader.Dispose()
                End If
            End If
            Me.disposed = True
        End If
```

```
    End Sub

    Public Sub Dispose() Implements IDisposable.Dispose
        Dispose(True)
        GC.SuppressFinalize(Me)
    End Sub

End Class
```

C#

```
public class Reader : IDisposable
{
    private StreamReader _reader;
    private bool disposed = false;

    public Reader(string filename)
    {
        this._reader = new StreamReader(filename);
    }

    protected virtual void Dispose(bool disposing)
    {
        if (!disposed)
        {
            if (disposing)
            {
                if (this._reader != null)
                {
                    this._reader.Dispose();
                }
            }

            disposed = true;
        }
    }

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }
}
```

Analisando o código acima, nota-se que existem dois métodos *Dispose*. O método que não tem nenhum parâmetro é o método fornecido pela *Interface IDisposable*. Além dele, criamos um método protegido (*protected*) e que pode ser sobrescrito nas classes derivadas que faz a análise dos recursos que devem ser descartados quando o método público *Dispose* é chamado.

Além disso, termos uma classe que implementa *IDisposable* é interessante porque podemos ainda utilizar em conjunto com o bloco *using*. Isso quer dizer que, ao utilizar o objeto dentro de um bloco *using*, o runtime se encarrega de invocar o método *Dispose* mesmo que alguma exceção ocorra até o término da operação e sem termos o código envolvido em blocos *Try/Catch/Finally*. Isso se dá porque o compilar de encarrega de transformar o bloco *using* em um bloco envolvido por *Try/Finally*.

Para finalizar, o código abaixo exibe o consumo da classe *Reader* que foi construída acima com o padrão *IDisposable*. A classe será utilizada em conjunto com o bloco *using*. Isso fará com que, ao chegar no término do bloco, o método *Dispose* é chamado automaticamente pelo runtime:

```
VB.NET
Using reader As New Reader("C:\Temp.txt")
    'executa outras operações
End Using

C#
using (Reader reader = new Reader("C:\\Temp.txt"))
{
    //executa outras operações
}
```

Casting

Efetuando conversões em tipos-referência

Uma das partes mais importantes do Common Language Runtime (CLR) é a segurança de tipos. Em runtime, o CRL sempre sabe qual tipo o objeto realmente é e, se em algum momento você quiser saber, basta chamar o método *GetType*.

Cast (ou conversão) é algo que os desenvolvedores utilizam imensamente na aplicação, convertendo um objeto em vários outros tipos diferentes (desde que suportados). Geralmente, o *cast* para o tipo base não exige nenhum tipo especial. Para citar um exemplo, você poderia fazer atribuir a uma variável do tipo **System.Object** a instância de uma classe *Cliente*, podendo utilizar o seguinte código:

```
VB.NET
Dim cliente As New Cliente()
cliente.Nome = "José Torres"

Dim obj As Object = cliente
```

```
C#
Cliente cliente = new Cliente();
cliente.Nome = "José Torres";

object obj = cliente;
```

Para fazer o inverso, ou seja, extrair da variável *obj* o Pedido que está dentro dela, é necessário efetuar o cast se estiver utilizando C#. O Visual Basic .NET não exige que você faça isso explicítamente, pois ele gera o código necessário para tal operação. No entanto é sempre menos performático fazer dessa forma e, além disso, dificulta encontrarmos possíveis problemas ainda em compile-time, ou seja, se o tipo não for compatível, somente descobriremos isso quando a aplicação for executada e uma exceção for atirada. Para melhorar isso no Visual Basic .NET, vá até as propriedades do projeto no *Solution Explorer*, em seguida na aba *Compile* e defina a opção *Option Strict* como ***On***. Isso obrigará o desenvolvedor a fazer as conversões explícitas no código. Para extrair o *pedido* da variável *obj*, o código é o seguinte:

```
VB.NET
Dim pedidoNovo As Pedido = DirectCast(obj, Pedido)

C#
Pedido pedidoNovo = (Pedido)obj;
```

O código acima somente resultará com sucesso se o valor contido dentro da variável *obj* for realmente um *Pedido*. Do contrário, a aplicação retornará um exceção do tipo **InvalidCastException**, informando que não é possível converter o que está em *obj* para *Pedido*.

Para evitar que problemas como este ocorra em tempo de execução, tanto o Visual Basic .NET quando o Visual C# fornecem alguns operadores que permitem que a conversão seja feita de forma "mais segura", evitando assim, o problema que identificamos acima. Felizmente para ambas as linguagens temos um operador que, se a conversão não for satisfeita, ao invés de atirar uma exceção, retornar uma valor nulo e, conseqüentemente, você deve tratar isso para dar continuidade na execução do código.

Estamos falando do operador **as** para o Visual C# e o operador **TryCast** para o Visual Basic .NET. Para exemplificar o uso destes operadores, vamos transformar o exemplo que fizemos logo acima para utilizar os operadores em questão:

```
VB.NET
Dim pedidoNovo As Pedido = TryCast(obj, Pedido)
If Not IsNothing(pedidoNovo) Then
    Console.WriteLine(pedidoNovo.Data)
```

```
End If

C#
Pedido pedidoNovo = obj as Pedido;
if(pedidoNovo != null)
{
    Console.WriteLine(pedidoNovo.Data);
}
```

É importante dizer que ao utilizar esses operadores, de nada vai adiantar se não verificar se está ou não nulo, pois se por algum motivo retornar nulo e você invocar algum membro deste objeto, o runtime atirará uma exceção do tipo **NullReferenceException**.